Using Skip Graphs for Increased NUMA Locality

Samuel Thomas, Roxana Hayne, Jonad Pulaj, Hammurabi Mendes
Mathematics and Computer Science
Davidson College
Davidson, NC, USA
{sathomas, anhayne, jopulaj, hamendes}@davidson.edu

***Abstract*—High-performance simulations and parallel frameworks often rely on highly scalable, concurrent data structures for system scalability. With an increased availability of NUMA architectures, we present a technique to promote NUMA-aware data parallelism inside a concurrent data structure, bringing significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Our architecture is based on a data-partitioned, concurrent skip graph indexed by thread-local sequential maps. We implemented maps and relaxed priority queues using such technique. Maps show up to 6x higher CAS locality, up to a 68.6% reduction on the number of remote CAS operations, and an increase from 88.3% to 99% on the CAS success rate compared to a control implementation (subject to the same optimizations, and implementation practices). Remote memory accesses are not only reduced in number, but the larger the NUMA distance between threads, the larger the reduction is. Relaxed priority queues implemented using our technique show similar scalability improvements, with provable reduction in contention and decrease in relaxation in one of our implementations.**

***Index Terms*—NUMA, concurrent data structures, skip graphs, locality**

I. Introduction

The increasing availability of computing cores on shared memory machines makes concurrent data structure design a critical factor for the design of high-performance applications or parallel systems. Non-blocking [1], linearizable ([2], a concept analogous to "serializable") structures are particularly appealing, since they can effectively replace sequential or blocking (lock-based) structures without compromising the semantics expected by users (systems designers). However, the design landscape for concurrent structures is changing: NUMA architectures emerge as a set of computing/memory "nodes" linked by an interconnect, making memory accesses within the same NUMA node cheaper than those made across different ones.

Under the usual assumption that threads are pinned to cores, we adopt the definition of *local memory* accesses as those operating in memory initially allocated by the current thread (under first-touch NUMA policy), and *remote* accesses as those accesses that are non-local. Note that our definitions are conservative, as data initially allocated by two different threads could indeed be located within the same NUMA node. Our goal is to increase NUMA *locality* – the ratio of local over total memory accesses, and research is very active in this area. Some approaches [3], [4], [5] focus on redesigning data structures with NUMA awareness, which is effective as we have full ability to exploit the structure's internal features for the task. Unfortunately, complete redesigns can pose significant development and research efforts, unsuitable for non-specialists. On the other hand, approaches such as [6] allow sequential structures to be "plugged-in" and benefit from NUMA-aware concurrency, based on replicating the dataset among nodes, batching local operations, and coordinating batches as to minimize inter-node traffic.

A critical goal in concurrent data structure design is the reduction of *contention* for synchronized memory accesses, characterized when two or more threads operate concurrently on nearby locations in memory (e.g., same cache line). Synchronized operations introduce memory fences in the cache-coherence protocol, and optionally provide enriched semantics, such as `get_and_increment()` (atomic increment) or `compare_and_swap()` (CAS) (conditional atomic exchange), so they are critical for lock-free data structure design. However, they also introduce high invalidation traffic in the cache-coherence system, particularly under contention. With NUMA, it is even more critical that contention is reduced, as such traffic happens *across* different memory domains, resulting in expensive access costs. Reducing contention can be attained by promoting *internal data parallelism* for synchronized memory accesses, and our technique simultaneously promotes a reduction on remote memory accesses on NUMA.

Our contributions. We present a technique to promote NUMA-aware data parallelism inside the concurrent data structure, bringing significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Our design is based on integrating thread-local sequential maps with skip graphs ([7], [8], described also in Sec. II), while performing a data partitioning scheme over the skip graphs for increased NUMA locality. By "qualitative" increase in NUMA locality, we mean that remote memory accesses are not only reduced in number, but the larger the distance between threads in the system, the larger the reduction is (Sec. V). At a high level, skip graphs can be viewed as multiple skip lists [9] that overlap, so we partition subcomponents of skip graphs among threads as to promote

2643-3001/20/$31.00 ©2020 IEEE
DOI 10.1109/SBAC-PAD49847.2020.00031

higher NUMA locality and reduced contention (increased data parallelism). We design internal algorithms to take full advantage of the partitioned dataset in order to promote this goal. Note that, as originally defined, skip graphs are expensive data structures, so our technique is made viable in practice by incorporating existing thread-local indexing and well-documented laziness principles [5], [10] into our design. As a proof-of-concept, we implemented maps and relaxed priority queues ([11], [12], [13], [14]). Maps have been implemented with and without using laziness techniques, but always using thread-local indexing similarly to [5]. We are competitive with state-of-the-art maps [5], [10], [15]: in some cases, we see 80% increased performance, while in others, we see similar performance to the faster running implementation. As part of our NUMA locality assessment, we observe a 6x higher CAS locality, a 68.6% reduction on the number of remote CAS operations, and an increase from 88.3% to 99% of CAS success rate when using a lazy skip graph map implementation, as compared to our *control* – a skip list subject to the same codebase, optimizations, and implementation practices. Memory access patterns are visualized on Sec. V, showing evident qualitative improvement.

We also contribute by implementing *relaxed priority queues* [11], [12], [13], [14], which return an element among the k smallest elements in a set, rather than the absolute smallest. We not only use our data partitioning technique, which brings increased NUMA locality and reduced contention, but we consider a couple of algorithmic variations that further harness key structural features of the skip graph. Additionally, we overview a *formal argument* (proved thoroughly in [16]) indicating that one of our two priority queue protocols is subject to smaller contention and it is also slightly less relaxed (that is, the removed elements are closer to the minimum element as defined in a strict priority queue). We proceed with an overview and background in Sec. II, and related work on Sec. III. Design and implementation are further discussed in Sec. IV, with evaluation in Sec. V and conclusion in Sec. VI.

II. Architecture Overview and Background

In this section, we start with an architectural overview, then we discuss skip graphs in more detail. We discuss our NUMA-aware optimizations (our data partitioning scheme), and provide a general description of an implementation variant of our mechanism. Section IV complements our discussion with further contributions more related to implementation rather than architecture.

General Architecture. Our overall architecture consists of an underlying skip graph [7], called a *shared structure*, and multiple thread-local, sequential, navigable maps called *local structures*, one per thread. The local structures allow insertions, removals, and contains operations to "jump" to positions in the shared structure near to where they will complete. The "jump" is done on thread-local memory, which contributes to reducing remote memory accesses since we avoid traversing long paths in the shared structure (distributed across multiple NUMA memory banks). Once in the shared structure, our data partitioning scheme over the skip graph promotes a further reduction in remote memory accesses due to our partitioning mechanism. We call our overall structure *layered structure* due to this general architecture.

We say *nodes* store *elements*, although we use these terms interchangeably. We further use the terms *local nodes* or *shared nodes* to refer to those nodes belonging to a local structure or to the shared structure, respectively. We now describe how the local and shared structures interact. When a thread inserts an element e, it first creates a shared node `s` in the skip graph, and then the thread's local structure will map $e \to$ `s`. When a thread removes an element e, it first (i) *logically delete*s the shared node `s` in the skip graph; which will cause the next two events, in arbitrary order: (ii-a) a "lazy" physical removal is done by traversing threads in the shared structure and (ii-b) a "lazy" physical removal of the local structure entry $e \to$ `s` is done by the thread that originally inserted the element.

Skip Graphs. Fig. 1 shows a skip graph, which performs the role of our shared structure. A skip graph is composed of many singly-linked lists across multiple levels 0...*MaxLevel*. Each level i has exactly 2^i linked lists, and partitions the nodes in level $i-1$ (for $i > 0$) in two sublists. In the level-0 list (called λ), all nodes are present. The two level-1 lists, 0 and 1, partition the level-0 list, and so on. In the original definition of skip graphs, aimed at peer-to-peer distributed applications, the specific partitioning is probabilistic. In this paper, we have a *partitioning scheme* that will allocate nodes as to maximize data locality in the operations in our overall data structure (we describe our partitioning scheme below).

Skip lists [9] are similar to skip graphs, but the former contain one linked list per level, and the later 2^i lists at level i. In a skip list, all elements belong to level 0, 1/2 of the elements at level 1, 1/4 of the elements at level 2, and so on. Hence, a skip graph is a collection of overlapping skip lists: in Fig. 1, if we select exactly one linked list per level, we obtain a skip list. In Fig. 1, one skip list is highlighted, which we denote $(\lambda, 0, 01)$ after the linked lists that must be chosen to define it, from the bottom level to the highest level. This way, 39 is in the skip list denoted by $(\lambda, 1, 10)$, and 85 is in the skip list denoted by $(\lambda, 0, 00)$. Skip graph searches are skip list searches: start from a node's top level, and follow high-level pointers as far as possible before moving down levels. For example, from 16 we reach 63 by following the path 16 $\to$ 39, $\downarrow$, 39 $\to$ 45, 45 $\to$ 62, $\downarrow$, 62 $\to$ 63. Note we only follow references from the skip list 16 (our starting point) belongs to in its top level: $(\lambda, 1, 10)$. We refer to any skip list within the skip graph as *shared skip list*, and any of the individual linked lists within the skip graph as a *shared linked list*.

Data Partitioning. Even though the skip graph is shared by all threads, we limit where each thread operates

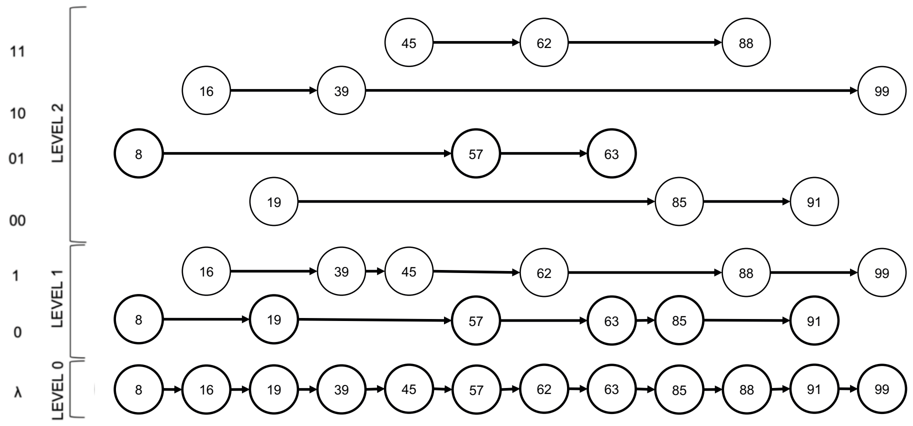

Fig. 1: A skip graph can be seen as a set of skip lists sharing their levels (one of them is highlighted). It contains 2^i linked lists at each level i. With T threads, we have $T/2^i$ of them working in each level-i linked list, separated by NUMA proximity, which increases data access parallelism, reduces contention, and increases NUMA locality.

on it, configuring a *partition scheme*. First, we establish that the maximum level of the skip graph is $\text{MaxLevel} = \lceil \log(T) \rceil - 1$, where T denotes the number of threads in the system (named $\{T_0 \dots T_{T-1}\}$). While such a "low" MaxLevel does not guarantee logarithmic searches by itself, our local structures perform the role of the missing higher levels as they "jump" to positions in the shared structure nearby where we expect operations to complete. We then divide the skip graph lists in the following way: (i) each thread T_i has a sequence of MaxLevel bits called a *membership vector*, denoted by M_i; (ii) T_i can only operate on the skip list characterized by the suffixes of M_i. That skip list is called the *associated skip list* of T_i, denoted L_i. For example, consider Fig. 1. Because $\text{MaxLevel} = 2$, T_i's membership vector M_i is a 2-bit string, say 01. Hence, T_i will always insert, remove, and search by following paths of the skip list $L_i = (\lambda, 0, 01)$. Note that since $\text{MaxLevel} = \lceil \log(T) \rceil - 1$, each top-level linked list is shared by 2 threads, and any arbitrary level-i list is operated by at most $T/2^i$ threads. An effective data partitioning is induced as we distribute threads over the skip graph as above, as we assume threads operate in different NUMA domains, and allocate memory within their domain. Furthermore, we have the opportunity to increase NUMA locality by having threads pinned to "closer" hardware to share more lists in the skip graph. We do that by generating membership vectors according to *physical NUMA features* of the machine. For example, consider a system with $T = 16$ threads ($\text{MaxLevel} = 3$) and 2 NUMA nodes, each with 2 CPUs, each of those with 2 cores, each of those with 2 hyperthreads. We will give the same membership vector for any two threads running on hyperthreads of the same core; membership vectors with a common 2-bit suffix if threads run on different cores, and with a common 1-bit suffix if threads run on different CPUs; and finally membership vectors with no common suffix if threads run on different NUMA nodes. Now, on the level-3 linked lists, any contention relates to core-local data (and hopefully located on the core's closest cache); on the level-2 linked lists, any contention relates to CPU-local data; on the level-1 lists, any contention relates to NUMA-local data. Not only we expect less contention in upper-level lists, because they are shared among less threads, but we expect this *contention to relate to more local data*. Further, any search that traverses the skip graph, as we discussed, is a skip list search. Hence, we *first* traverse core-local data, and *if we go down a level, the target data cannot be located in the same core*. Similarly, we then traverse CPU-local data, then NUMA-local data, and if we ever go down a level, the target data must be found remotely, where local/remote depend on the level in question. If we ever leave, say, a NUMA node, we never come back to it. The same applies to CPU-local data, or core-local data, and this happens **as a direct consequence of our data partitioning mechanism, and our choice of data structure**. We have an automated mechanism that generates membership vectors based on inspecting the system's CPU/socket/domain structure.

Alternative shared structure. In order to further explore benefits and tradeoffs of skip graphs, we also created and tested a second shared structure, called a *sparse skip graph*. This structure is a skip graph where elements are made present in level i *of any shared skip list* with expectation $1/2^i$, just like in a regular skip list. The combination of skip graph partitioning and skip list refinement makes elements be present in level i *of a particular linked list* with expectation $1/4^i$. For instance, in Fig. 1, each of the level-1 lists "0" and "1" would partition only 50% of the elements of "λ", which would be selected at 50% chance independently. So, "0" and "1" would each have about 25% of the elements in "λ". Similarly, the level-2 lists "00" and "01" would partition only 50% of the elements of "0", each selected at 50% chance, independently. So, lists "00" and "01" would each have about 6.25% of the elements in "λ". With sparse skip graphs, only elements that reach the top level are added to the local structures. This is crucial because the local structures, besides pointing to shared nodes nearby the target destinations, should also point to maximum-level nodes from which we can start an efficient search. Hence, using sparse skip graphs gives two immediate advantages: (i) the local structures are smaller; and (ii) the insertion and removal in the shared structure requires changes in less than MaxLevel levels. The tradeoff is that the starting point given by the local structures is not as close to the requested element compared to regular skip graphs.

III. Related Work

Skip Lists and Skip Graphs. Skip lists first appeared in [17], although [18], [19], [20] were most widely discussed in the literature [21]. Skip lists have also been used to implement priority queues, either exact [22], [23], [24] or with a relaxed definition [11], [12], [13], [14]. SkipNets [8] are similar (if not identical) to skip graphs, proposed relatively at the same time. We consider those equivalent, and equally applicable. Skip graph variations, such as in [25], typically address issues related to distributed systems, such as node size; we are aware of a single concurrent implementation in shared memory in [26], although it is lock-based, in contrast with both of our lock-free variants. Our implementation relies heavily on laziness, as we postpone much of the internal work until they are absolutely needed. The "No Hotspot" skip list [10] uses similar lazy principles, albeit with a different protocol. The "Rotating" skip list has a novel construction ("wheels") meant to improve cache efficiency and locality, and also constitutes a modern, state-of-the-art implementation.

NUMA awareness and layered design. The work presented in [27] gives a systematic approach to provide NUMA-awareness to locks. Tailor-made data structures for NUMA systems, such as [3], [4] have also been developed, using (now) standard techniques such as elimination [28] and delegation [29]. We think that "blackbox" approaches, such as in [6], are interesting as they relieve systems programmers from "customizing" their data structures for NUMA, a notoriously complicated task for non-specialists (and specialists alike [21]). NUMASK [5] is an interesting skip list that uses its higher levels as a hierarchical "index" to the bottom-level list, which stores the dataset. In our case, the dataset is located in a structure of its own, a multi-level skip graph. This allows for our data partitioning mechanism, designed to (i) reduce non-local NUMA traffic, and particularly avoid traversals that navigate back and forth across NUMA nodes; and (ii) reduce contention by creating areas within the shared structure where only subsets of threads operate. Our thread-local indexing, similarly, is more detached from the dataset, and could be implemented with any sequential, navigable map. In our implementation, for example, our map is actually a combination of a search tree and a hash table. Finally, our indexes are not replicated, but partitioned. Even with our optional load balancing mechanism in place, threads donate nodes but do not replicate their indexing. Apart from differences of granularity and function, the idea of separating thread-local views and shared views has been seen before in [30], although their approach is more akin to combining, as they eventually merge thread-local views into the shared structure from time to time.

A **brief announcement** (i.e. not a full paper) related to this work was published in [31], and substantial new material has been developed since then, including: lazy skip graphs (p. 4), load balancing (p. 5), the commission period policy (p. 5), relaxed priority queue algorithms (p. 5) and their analysis (p. 6). This paper is self-contained and includes all the new material since the brief announcement in [31].

IV. Implementation Details

Section II gives a design overview, and this section gives more insight into implementation and correctness, and discusses optimizations related to laziness and physical removal. We will discuss first the implementation of "map" abstract data types (ADTs), and later highlight the differences in the relaxed priority queue protocols.

Laziness. State-of-the-art concurrent data structures rely heavily on postponing internal work until they become absolutely needed, in the hope that they become unnecessary. We implemented a *lazy* variant of our layered structure, employing this principle as: (i) The insertion of shared nodes in the skip graph is done in the level 0 first, and we only complete the insertion at upper levels when the node in question is requested to start a search operation. (ii) Removals are performed logically by "invalidating" a shared node, and nodes are marked for physical removal only when the threads that originally inserted those nodes find them invalidated after a minimal *commission period* for which they exist in the structure. As the physical removal of a node is expensive, the commission period is intended to have this operation done only when necessary. Experimentally (Sec. V), we found that a commission period proportional to the number of threads, say $350000 \cdot T$ cycles, for instance, performs *very well* under high-contention without introducing too much overhead in low-contention (in the latter, a longer commission period could leave the data structure much larger at times). (iii) We implemented an optimization that removes *chains* of marked shared nodes with a single CAS operation, and, related to laziness, we do that only when *substituting* a chain of marked shared nodes with an inserting node. Although this protocol has the potential to leave too many marked nodes in the data structure, we verified experimentally that the number of traversed shared nodes per operation is less than in a skip list up to 96 threads.

Physical removal. We now expand our discussion of valid and invalid nodes to a more formal definition. Each node has a *valid* bit and a *marked* bit in each successor reference in the skip graph. When a node's level-0 reference `marked` (resp. `valid`) bit is set, we say the shared node is *marked* (resp. *invalid*). The concepts of *unmarked* and *valid* are obvious. In the non-lazy version, we do not use the `valid` bit, and in that case an unmarked shared node indicates presence in the abstract key set, while a marked shared node indicates a *logically deleted* node.

In the lazy version, an unmarked, valid node indicates presence in the abstract set; an unmarked, invalid node indicates absence from the abstract set (logically deleted), but also that the process of physically unlinking the node has not started; a marked node can only be invalid, and in that case the process of physically unlinking is ready to start. Each shared node `s` has a field `s.allocTimestamp`, set at shared node construction, used to calculate when a node's commission period has elapsed, thus making it a candidate for physical removal (see the discussion on laziness, above). The removal process happens in distinct phases. (i) A thread that traverses the data structure will mark a shared node with an expired commission period only if such node has been inserted by the thread in question (recall that "marking" nodes means marking its successor reference at level 0). (ii) When threads remove marked nodes from their local structures, they also mark upper-level successor references in order to promote physical removal of the shared node in the upper levels as well. As in many cases logically deleted nodes have not been inserted in upper levels, this additional reference marking is not always necessary or complete. (iii) Finally, traversing threads perform the physical cleanup using the relink optimization discussed below. Note that (i) and (ii) are synchronization operations done in *local memory*, and we consider this protocol for maximal locality as one of our implementation-related contributions.

In textbook skip lists [21], we indicate willingness to physically remove a node `s` by marking its `s.next[i]` references for all levels $i \dots 0$ the node belongs to. Within a level i, searches performed on behalf of insertions and removals physically remove nodes with marked `next[i]` references by employing a *single* CAS per node. In contrast, both our skip lists or skip graphs remove *sequences* of marked references with a single CAS per sequence, a trivial optimization that we we denote by *relink optimization.* The correctness of this protocol is trivial when we consider that marked references are immutable. Our technical report[16] describes the algorithms in detail and discusses how they are lock-free and linearizable.

Unbalanced workloads. We have an optional mechanism for handling unbalanced workloads, which addresses the following scenarios: (i) Some threads may only insert, while others only perform removals/contains. (ii) Distinct groups of threads may insert in distinct partitions of the element space. Both scenarios are problematic because threads do not necessarily find good starting points for their search operations if their local structures are empty or skewed towards a partition of the element space. Our load-balancing mechanism is based on having threads donate a fraction of their nodes inserted in the shared structure so they are added to local structures of other threads. A background thread takes into account the number of inserted elements announced by every worker thread, and, based on those numbers, continuously indicates to each worker thread the fraction of inserted nodes that are requested for donation. The worker threads place such fraction of inserted nodes into donation queues (one per worker thread), which are collected by the background thread and distributed uniformly among all other worker threads. Threads inspect receiving queues for incoming nodes, and add them into their local structures. Specifically, if a worker thread T_i announces the insertion of I_i elements out of a total of $I_T = \sum_{\{0 \le i < T\}} I_i$, define $q_i = I_i / I_T$. If $q_i \le 1/T$, then T_i donates the fraction q_i of its elements; otherwise, it donates $(Tq_i + q_i - 1)/Tq_i$ of them. Donated nodes have been inserted in their bottom-level by a thread T_i, while they might be inserted in upper levels by a thread $T_j, j \neq i$. We are currently working on delegating the creation of upper levels to the original thread T_i, avoiding to cross NUMA nodes when building up those levels. On lazy implementations, with levels built only when needed, the impact of this problem is reduced.

Priority Queues. Our layered structure can implement priority queue ADTs (and relaxed versions of it) in addition to sets/maps. Similarly to [23], [24], [22], we rely on marking elements in the bottom level of the (now) skip graph in order to logically delete elements. We consider two *relaxed priority queue* approaches [11], [12], [13]: (i) the use of the spraying technique of [14] over skip graphs; and (ii) a custom protocol that traverses the skip graph deterministically, marking elements along this traversal. We consider (ii) as one of the contributions of this paper.

Regarding option (i), the main idea of [14] is to disperse threads over the skip list bottom level through a random walk called a *spray.* If we apply the technique to a skip graph, each thread T_i would navigate only through its associated skip list L_i. We see several advantages of performing spray operations over skip graphs rather than skip lists. Our partitioning scheme will still incur in better memory locality and reduced contention, as discussed before. On the other hand, we show that spraying over skip graphs has a slightly bigger removal range (same asymptotically, but higher nevertheless, proved in [16]). Related to option (ii), we implement a deterministic traversal in the skip graph, marking elements along the way. Informally, a thread T_i starts at the highest level of its associated skip list, traverses marked nodes, and attempts to mark at the current position. If the attempt succeeds, a node has been logically deleted, otherwise the thread moves down a level, traverses marked nodes, and proceeds similarly. At level 0, two mark attempts are tried, and upon failing the second one, the process is restarted [16]. We prove on [16] that the number of CAS operations required to logically delete a sequence of T nodes in our deterministic protocol is $2T$ (so each node is subject to contention by 2 threads in expectation). In either approach, our layered structure gives the opportunity to perform physical cleanup on local structures using *combining* [32], [33]: as we always remove a *whole prefix* of the local structure containing logically deleted nodes, many nodes could be removed at once, at cost comparable to a single removal.

PQ Formal Arguments. We overview our formal arguments that quantify (A) the removal range of spraying operations in skip lists and skip graphs; and (B) the contention anticipated for approaches (i) and (ii) discussed above, over skip graphs. In [16] we provide the complete formal arguments for the theorems below. We first calculate the removal range of spray operations in skip graphs:

Theorem 1 ([16]). *For each* $\mathtt{SPRAY}(\log \mathtt{T} - \mathtt{1}, \log \mathtt{T}, \mathtt{1})_{\mathtt{j}}$*, for any list j in the maximum level of a perfect skip graph, the position of the node on which the operation lands is at most* $\frac{T}{2} + \log T \cdot (T-1) - 1$.

Compared to spraying over skip lists, the larger bound is related to the fact that skip graphs have multiple starting points offset from each other. We show, however, that the probability of a skip graph spray to reach an arbitrary node in the bottom level is $\leq 1/T$:

Theorem 2 ([16]). *Fix* $\mathtt{H} = \log \mathtt{T} - \mathtt{1}$*,* $\mathtt{L} = \log \mathtt{T}$*, and* $\mathtt{D} = \mathtt{1}$*. Then for any position x, the probability that the spray operation lands on x is* $\leq \frac{1}{T}$.

In [16], we prove that one of our relaxed priority queue algorithms (which we call SGMARK), designed precisely in order to exploit the central structural features of the skip graph, can remove elements from a range of T elements with proven contention being exactly 2 *for any number of threads T*:

Theorem 3 ([16]). `SGMARK` *on a perfect skip graph ensures that exactly $T = 2^n$ nodes are marked. Furthermore, 2 threads contend for each of the first $T-1$ consecutive nodes, and 1 thread tries to CAS (logically delete) the last node.*

V. Evaluation

We performed experiments in a system with 2 Intel Xeon Platinum 8275CL CPUs, each with 24 cores running at 3.0GHz (96 hardware threads total). The system has 192GB of memory and two NUMA nodes. The NUMA-distance tool `numactl --hardware` reports intra-node distances of 10 and inter-node distances of 21 (not an issue here as we demonstrate NUMA locality using software instrumentation). The system runs Ubuntu Linux 18.04 LTS with kernel 4.15.0. We compile tests with `g++ -std=c++11 -O3 -m64 -fno-strict-aliasing`.

Experiment setup. We report the *total* number of operations per millisecond achieved in trials having from 2 up to 96 threads. Each trial is an average of 5 runs of 10s each, and follows exactly the testing procedure of Synchrobench [34] with the flag `-f 1`. This flag indicates that the testing procedure tries to match each trial's requested percent of *update operations* (inserted and remove) as much as possible, and that only successful inserts or removals count as update operations. The testing procedure, as well as random number generation, are identical to Synchrobench. We run a *read-heavy* (RH) load, with a requested 20% of update operations, a *write-heavy* (WH) load, with a requested 50% of update operations, and a *priority queue* (PQ) load, with 50% of insertions and 50% of removals, all distributed uniformly at random across all threads (except for our load-balancing tests). If X% of operations correspond to *successful* updates[1] in a given experiment, we say we had X% of *effective updates*, and we report that percentage in each associated graphic caption. Our experiments are defined to be *high contention* (HC) when the key space is 2^8, *medium contention* (MC) when it is 2^{11}, and *low contention* (LC) when it is 2^{17}. The structures are preloaded with 20% of their maximum capacity before any measurements, except for the LC tests, which are preloaded with 2.5%. LC tests and analysis are presented in [16]; here we will focus on WH-HC and WH-MC tests. Threads are pinned to each CPU, and we fill a socket before adding threads to another socket. We allocate memory with `libnuma`, in chunks capable of holding 2^{20} objects, in order to amortize the expensive cost of `numa_alloc_local()`. Membership vectors are generated as described in Sec. II, and we obtain data from `/proc/cpuinfo` on Linux to automatically number and pin threads.

1 - Performance results. Figures 2 and 3 show write-heavy (WH) results for the HC and MC contention scenarios. Low-contention results (LC) and read-heavy (RH) results are presented in [16]. In our graphs, `layered_map_{sg,ssg}` refers to using C++ `std::map` in conjunction with the hash [35] as local structures, respectively over regular or sparse skip graphs (p. 4) as shared structures; `lazy_layered_sg` is the lazy variant of `layered_map_sg`; `rotating` is [15], `nohotspot` is [10], and `numask` is [5] as found in Synchrobench's GitHub (mid August 2019). For the purpose of isolating individual design components in our analysis, we also developed as **control**: a locked skip list; a concurrent skip list with the same codebase and practices as our skip graph code, including our relink optimization (p. 5); a skip graph without layering; and finally our layered design (i) over a linked list (`layered_map_ll`) and (ii) over a skip list (`layered_map_sl`). The former is essentially a `layered_map_sg` with maximum level 0, and the latter a `layered_map_ssg` with a single constituent skip list (hence, with no opportunity to implement our partitioning

[1]Failed inserts due to pre-existing keys, or failed removals due to absent keys are essentially "contains" operations, as they both return immediately after identifying the respective scenarios above.

scheme). Non-layered skip lists or skip graphs have maximum level x if the test's key space is of size 2^x, and layered versions follow our partitioning scheme definitions (p. 3).

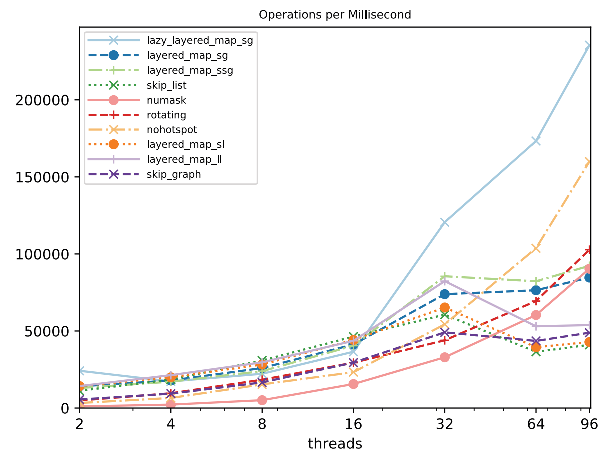

Fig. 2: HC, WH: 32% effective updates.

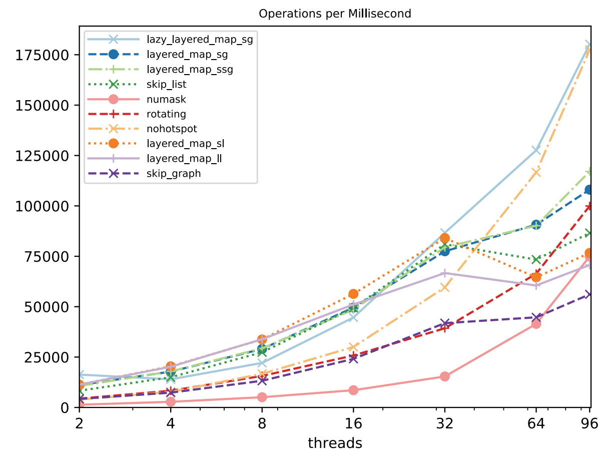

Fig. 3: MC, WH: 32% effective updates.

With a small key space (HC-WH), `layered_map_ll` performs better than `layered_map_sg` and `layered_map_sl` up to 32 threads, but the performance degrades quickly as we have more threads or the key space gets bigger (MC-WH, Fig. 3; LC-WH, [16]). The reason is that with more threads or bigger key spaces, more elements need to be traversed in the unique linked list upon searches. Then, we could be tempted to say that the multilevel shared structure in `layered_map_sg` is the reason it performs better in MC-WH, but note that `layered_map_sl` performs similarly than `layered_map_ll` in the same case MC-WH for high thread counts. The *reason* why `layered_map_sg` performs better in the MC-WH scenario is, therefore, the unique differentiating factor: the partitioning scheme in the shared structure (the skip graph). Further, in the same MC-WH scenario, we note a clear performance separation between `layered_map_ssg` and `layered_map_sl` after 32 threads. The unique differentiating factor here is multiple vs one skip list as a shared structure. So, the existence of multiple, overlapping of skip lists, employing a partitioning scheme across threads is the differentiating scalability factor for the good performance of our `layered_map_sg`.

As far as the lazy implementation performance, we see it as a combination of (i) the effectiveness of our partition scheme for increasing NUMA locality and reducing contention (implied above and *verified* below, in item #2); (ii) the commission policy to unlink invalid, marked nodes (isolated right below); and (iii) the fact that with smaller key spaces, threads will more commonly find unmarked nodes through their local hashtable, which performs much better compared to the `std::map` local structure. We show a `lazy_layered_map_sgNA` where we make the commission period zero, as control. Under HC-WH, [15] performs well, and our control implementation is comparable to [5], [10]. Under MC-WH, [10] performs well, and our control implementation is comparable to [5], [15]. In any case, we confirm our expectation that naive skip graphs scale poorly, because while in a skip list the expected number of levels of each node is 2, in a skip graph it is always the maximum. Further, on Tbl I, we see how `layered_map_sg`, without any commission period, requires a *lot* more CASes per operation than other structures. With that in mind, and considering our indications that the partitioning scheme works, we have *first* to make sure that skip graphs become *viable* with techniques such as lazy insertions/removals, the commission period, and our relink optimizations mentioned in p. IV.

2 - NUMA locality and contention reduction. We *verify* that our partitioning mechanism promotes better NUMA locality and reduced contention below. Figures 4 and 5 show heatmaps where coordinates (i, j) indicate the distribution of CAS instructions per operation performed by thread i into a node allocated by thread j, instrumented manually on node access functions on the 96-thread MC-WH scenario. The memory access pattern shows that the larger the distance between thread IDs (which we adjusted to match the NUMA distance), the smaller the number of memory accesses. This access pattern correlates with physical NUMA distance since (i) our testing framework assigns threads pinned to closeby locations (considering hyperthreads, cores, CPUs, NUMA domains) with closeby IDs; and (ii) our partitioning scheme uses these reassigned thread IDs in order to partition the skip graph.

Comparing the lazy skip graph and a skip list (which serves as our *control*, as it has been implemented using the same codebase and practices), the heatmaps indicate a *dramatic* increase in CAS NUMA locality. We provide graphics for other structures (non-lazy skip graph, non-lazy sparse skip graph) in [16]. Also related to locality, we expect fewer cache misses due to our partitioning mechanism, which we confirm (yet document in [16]): with

32 threads, our layered skip graph has a reduction of 21% in L1 misses, 41% in L2 misses, and 18% in L3 misses.

Related to contention, Tbl. I shows additional metrics collected via manual code instrumentation, on the 96-thread HC-WH scenario. Both our heatmaps and Tbl I *do not count* CAS/read/write operations performed over an inserting node, otherwise locality would be artificially inflated with operations that are inherently local, as threads have to initialize their allocated nodes. Any CAS operation metric presented is a *maintenance CAS*: an operation required to link, unlink, or cleanup nodes.

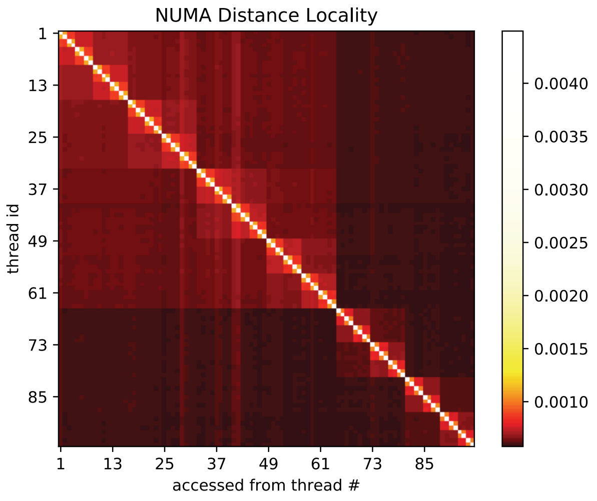

Fig. 4: MC-WH CAS heatmap, lazy map/SG.

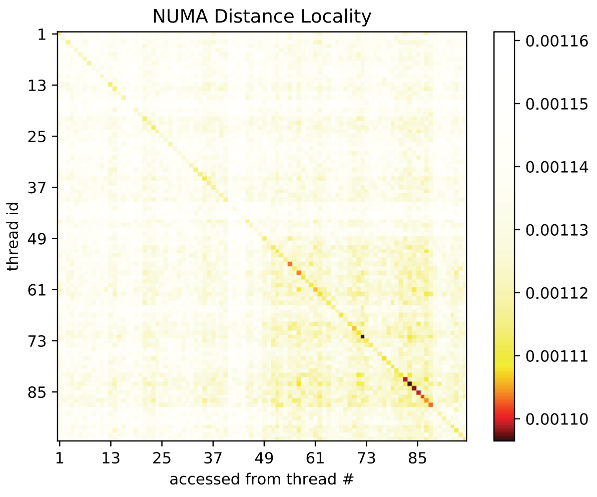

Fig. 5: MC-WH CAS heatmap, skip list.

Although `lazy_layered_map_sg` performs slightly more reads per operation than skip lists, it performs 68% less remote maintenance CASes per operation. The CAS success rate is substantially higher (99% in `lazy_layered_map_sg` vs. 88.3% in the skip list). Both the increase in NUMA

	lazy map/sg	map/sg	map/ssg	skip list
loc. reads/op	9.105	8.933	4.264	0.477
rem. reads/op	48.076	54.521	65.123	45.392
loc. CAS/op	0.02508	0.177	0.0137	0.012
rem. CAS/op	0.3493	2.524	0.888	1.113
CAS succ. rate	0.999	0.986	0.982	0.883

TABLE I: 96 threads, HC-WH. CAS/op does not include uncontended CAS operations upon node insertion. Comparing the lazy map/sg with the skip list, we can observe 6x more CAS locality, 65% less CAS/operation, and substantially better CAS success rate.

locality, discussed before, and the contention reduction, just discussed, are attributed to our *partitioning scheme*, designed precisely with those goals in mind (p. 3). Our technical report [16] presents similar results for atomic reads. Atomic writes are only used to initialize nodes before insertion, so these operations are all contention-free and 100% local, so they are not measured.

3 - Relaxed priority queues. Figure 6 tests multiple implementations for relaxed priority queues using skip graphs. The `spray` implementation consists on the application of the spraying technique of [14] over skip graphs; the `sg_spray` implementation is our custom protocol that traverses the skip graph deterministically, marking elements along this traversal (described in detail in [16]). For both approaches, we also show lazy variants (`spray_lazy` and `sg_spray_lazy`), using the ideas presented earlier in the context of map ADTs. We also have a *control* skip list (implemented using the same codebase and practices) where we perform spray operations as in [14].

Lazy versions are expected to perform faster for similar reasons they perform better in maps, although, for relaxed priority queues, we cannot make guarantees about their degree of relaxation. The reason is that upper-level lists are substantially more sparse in our lazy implementations, and elements are added to those lists by demand, something which we cannot (at this point) reasonably model. Now focusing on the non-lazy versions, we note that `spray` scales better than `sg_spray`. Although Theorem 3 indicate that `sg_spray` is subject to a very small contention, we also show in [16] that the range of `spray` is slightly larger. Fig. 7 shows an experiment similar to the one in [14], where we perform traversals and only note which nodes would be marked, without actually marking any element. The experiment shows that `sg_spray` is indeed less relaxed than `spray`, so the reduced scalability of `sg_spray` is explained in Fig. 6. In conjunction, Figures 6 and 7 essentially exhibit a tradeoff between priority queue relaxation and scalability.

4 - Load balancing. In order to evaluate load balancing, we consider two *extremal* experiments: (i) A scenario where only thread T_0 inserts, and all others perform removals and contains operations in an MC-WH experiment, Figure 8 still shows that the *sizes* of the local structures are similar. In that figure, the X-axis represent

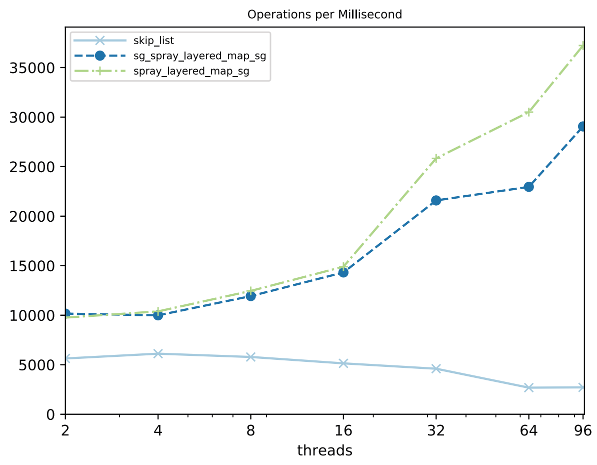

Fig. 6: MC-PQ: 82% effective updates.

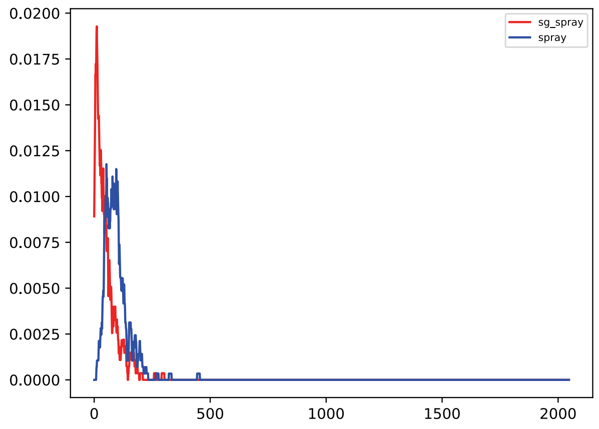

Fig. 7: MC-PQ: Key range of removed elements (%).

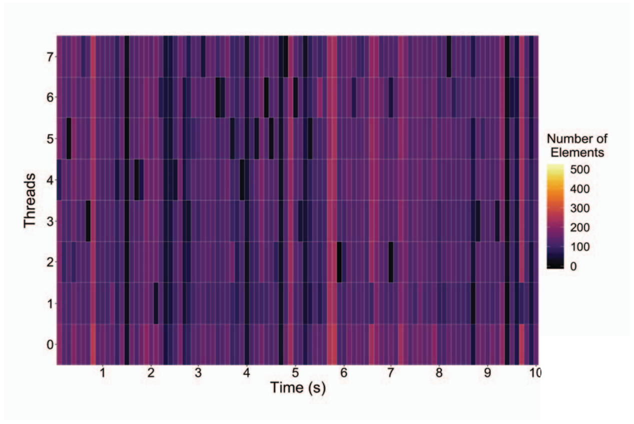

Fig. 8: Local structure sizes, keyspace = 2048. The color indicates the number of elements per thread, per time.

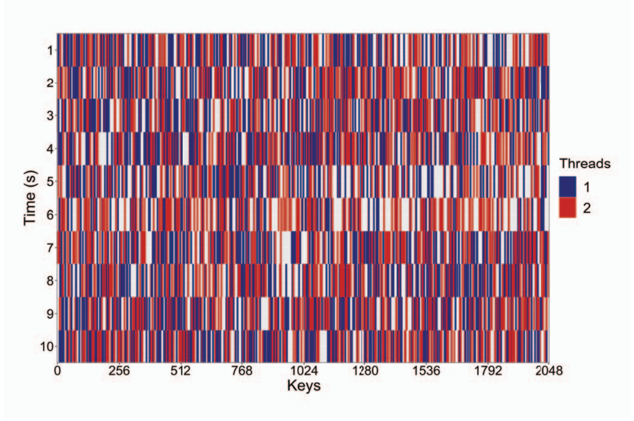

Fig. 9: Key distribution, 2048 keyspace. The color indicates the owning thread, per key, per time.

100 discrete time points, varying from 0-10s, the Y-axis represents 8 different threads, and the color indicates the local structure size (not counting marked elements). (ii) A scenario where 1/2 of the threads insert only on the 0-1023 range of a 2048 keyspace, and 1/2 of the threads insert only on the 1024-2047 range of the keyspace. Figure 9 shows that the key distribution of two threads that belong to two different groups are spread throughout the *whole* element space. In the figure, the X-axis represents possible keys, the Y-axis represents 10 discrete time points, one per second, and the color represents whether a particular key belongs to thread 1 or 2 in their local structure. Our load balancing mechanism has shown to have about 20% of scalability impact at the highest thread count, but it can be turned off completely in case the application has a uniform operation distribution among threads.

VI. Conclusion

We presented a technique to promote NUMA-aware data parallelism inside the concurrent data structure, bringing significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Our design is based on integrating thread-local sequential maps with skip graphs, while performing a data partitioning scheme over the skip graphs for increased NUMA locality. By "qualitative" increase in NUMA locality, we mean that remote memory accesses are not only reduced in number, but the larger the distance between threads in the system, the larger the reduction is. We provide an optional load-balancing mechanism for applications where the types of operation are not uniformly distributed among threads.

For relaxed priority queues, we consider two alternative implementations: (a) using "spraying", a well-known random-walk technique usually performed over skip lists, but now performed over skip graphs; and (b) a custom protocol that traverses the skip graph deterministically, marking elements along this traversal. We provide formal arguments indicating that the second approach is slightly more *relaxed*, that is, that the span of removed keys is larger, yet shows smaller contention and higher scalability.

References

[1] M. Herlihy and N. Shavit, "On the nature of progress," in *Principles of Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 313–328.

[2] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transaction on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[3] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "Cphash: A cache-partitioned hash table," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 319–320.

[4] I. Calciu, J. Gottschlich, and M. Herlihy, "Using elimination and delegation to implement a scalable NUMA-friendly stack," in *5th USENIX Workshop on Hot Topics in Parallelism*. San Jose, CA: USENIX, 2013.

[5] H. Daly, A. Hassan, M. F. Spear, and R. Palmieri, "NUMASK: High Performance Scalable Skip List for NUMA," in *32nd International Symposium on Distributed Computing (DISC 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 121, Dagstuhl, Germany, 2018, pp. 18:1–18:19.

[6] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for NUMA architectures," *SIGPLAN Not.*, vol. 52, no. 4, pp. 207–221, Apr. 2017.

[7] J. Aspnes and G. Shah, "Skip graphs," *ACM Transactions on Algorithms*, vol. 3, no. 4, Nov. 2007.

[8] N. J. A, D. Michael, B. Jones, and S. Saroiu, "Skipnet: A scalable overlay network with practical locality properties," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 2003.

[9] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," in *Workshop on Algorithms and Data Structures*, 1989, pp. 437–449. [Online]. Available: citeseer.ist.psu.edu/pugh90skip.html

[10] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 196–205.

[11] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 317–328. [Online]. Available: https://doi.org/10.1145/2429069.2429109

[12] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data structures for task-based priority scheduling," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 379–380. [Online]. Available: https://doi.org/10.1145/2555243.2555278

[13] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 277–278.

[14] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proc. of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 11–20.

[15] I. Dick, A. Fekete, and V. Gramoli, "A skip list for multicore," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, p. e3876, 2017.

[16] S. Thomas, R. Hayne, J. Pulaj, and H. Mendes, "Layering data structures over skip graphs for increased NUMA locality," *CoRR*, vol. abs/1902.06891, 2019. [Online]. Available: http://arxiv.org/abs/1902.06891

[17] W. Pugh, "Concurrent maintenance of skip lists," University of Maryland at College Park, Tech. Rep., 1990.

[18] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," in *Structural Information and Communication Complexity*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 124–138.

[19] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit, "A provably correct scalable skiplist (brief announcement)," in *Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006)*, 2006.

[20] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '04. New York, NY, USA: ACM, 2004, pp. 50–59.

[21] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[22] I. Calciu, H. Mendes, and M. Herlihy, "The adaptive priority queue with elimination and combining," in *Distributed Computing*, ser. Lec. Not. in Computer Science, F. Kuhn, Ed. Springer Berlin / Heidelberg, October 2014, vol. 8784, pp. 406–420.

[23] N. Shavit and I. Lotan, "Skiplist-based concurrent priority queues," in *IEEE International Symposium on Parallel and Distributed Processing*, 2000, pp. 263 –268.

[24] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," in *IEEE International Symposium on Parallel and Distributed Processing*, april 2003, p. 11 pp.

[25] M. T. Goodrich, M. J. Nelson, and J. Z. Sun, "The rainbow skip graph: A fault-tolerant constant-degree distributed data structure," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, ser. SODA '06. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006, pp. 384–393.

[26] H. Mendes and C. G. Fernandes, "A concurrent implementation of skip graphs," *Electronic Notes in Discrete Mathematics*, vol. 35, pp. 263–268, 2009.

[27] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *SIGPLAN Not.*, vol. 47, no. 8, pp. 247–256, Feb. 2012.

[28] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 206–215.

[29] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir, "Message passing or shared memory: Evaluating the delegation abstraction for multicores," in *Principles of Distributed Systems*, R. Baldoni, N. Nisse, and M. van Steen, Eds. Springer International Publishing, 2013, pp. 83–97.

[30] D. Akkoorath, J. Brandão, A. Bieniusa, and C. Baquero, "Global-local view: Scalable consistency for concurrent data types," in *Euro-Par 2018: Parallel Processing*. Cham: Springer International Publishing, 2018, pp. 492–504.

[31] S. Thomas and H. Mendes, "Brief announcement: Layering data structures over skip graphs for increased numa locality," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 422–424.

[32] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 355–364.

[33] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 257–266.

[34] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 1–10.

[35] M. Ankerl. [Online]. Available: github.com/martinus/robin-hood-hashing